Un ordonnanceur flexible pour machines multiprocesseurs hiérarchiques

Samuel Thibault

To cite this version:

Samuel Thibault. Un ordonnanceur flexible pour machines multiprocesseurs hiérarchiques. 16ème Rencontres Francophones du Parallélisme, ACM/ASF - École des Mines de Nantes, Apr 2005, Le Croisic, France. inria-00000137

HAL Id: inria-00000137
https://hal.inria.fr/inria-00000137

Submitted on 27 Jun 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants
Le Croisic, France, 5 au 8 avril 2005

inria-00000137, version 1 - 27 Jun 2005

Un ordonnanceur flexible pour machines multiprocesseurs hiérarchiques

Samuel Thibault

LaBRI, Domaine Universitaire, 351 cours de la Libération, 33405 TALENCE CEDEX - France
E-Mail : samuel.thibault@labri.fr

Résumé
L'évolution des machines multiprocesseurs vers des architectures de plus en plus hiérarchiques impose, pour en tirer la quintessence, de répartir les flots d'exécution et les données avec une extrême précaution afin de réduire au maximum les accès mémoire non locaux. Les bibliothèques de multi-threading actuelles fournissent très peu de fonctionnalités pour exprimer des directives de répartition au niveau applicatif, ce qui contraint les programmeurs à effectuer cette répartition explicitement en fonction de l'architecture sous-jacente, et donc de manière non portable. Dans cet article nous présentons : (1) un modèle permettant au programme d'exprimer dynamiquement la structure du calcul ; (2) un ordonnanceur capable d'interpréter cette modélisation afin de prendre de judicieuses décisions de placement hiérarchisé ; (3) une implémentation au sein de la bibliothèque de threads utilisateur MARCEL. Une expérimentation a été menée sur une application scientifique exécutée par une machine ccNUMA BULL NOVASCALE à 16 processeurs INTEL ITANIUM II ; les résultats obtenus montrent un gain de 50 % par rapport à un ordonnanceur classique et sont comparables à ceux que l'on obtient en effectuant le placement « à la main », ce qui n'est pas portable.

Mots-clés : threads, ordonnanceur, NUMA, SMP, Multi-Core

1. Introduction

« Désactivez l'hyperthreading ! » C'est malheureusement la réponse pragmatique récurrente face aux problèmes de performances rencontrés sur des serveurs équipés de processeurs hyperthreadés tels que les INTEL XEON. Cette situation est préoccupante car depuis quelques années on constate une croissance de la hiérarchisation — et donc une complexification — de l'architecture des serveurs de calcul contemporains (SUN WILDFIRE [1], SGI ORIGIN [2], BULL NOVASCALE [3] notament).
Ces machines, véritables « poupées russes », imbriquent les technologies permettant l'exécution simultanée de plusieurs threads au cœur d'un même processeur (SMT : *Simultaneous Multi-Threading*), le partage de mémoire cache par plusieurs de ces processeurs (puce multi-core) et l'intégration de cartes multi-processeurs (SMP) autour d'un réseau de type *crossbar*. Le tout constitue un serveur de calcul de type NUMA (*Non Uniform Memory Access*) où le temps d'accès à la mémoire dépend de la position du banc mémoire relativement au processeur y accédant (appelé facteur NUMA).
L'intégration récente des technologies SMT et multi-core complexifie donc notablement la structure des machines NUMA, machines que les systèmes d'exploitation avaient déjà des difficultés à exploiter efficacement. Hennessy et Paterson le relèvent pertinemment [4] à propos des systèmes proposés pour SGI ORIGIN et SUN WILDFIRE : *« There is a long history of software lagging behind on massively parallel processors, possibly because the software problems are much harder. »* L'introduction de nouvelles technologies matérielles confirme le besoin de développement logiciel et notre propos est d'apporter une solution *portable* visant à améliorer l'efficacité des applications multithreadées hautes performances sur les calculateurs modernes.
Exploiter ces machines de manière optimale est donc un vrai défi. Sans information sur l'affinité entre différentes tâches, on ne peut en effet pas prendre de bonnes décisions telles que regrouper sur un même nœud NUMA les tâches travaillant sur les mêmes données par exemple. La détection automatique de telles affinités est ardue, il est *a priori* plus simple que l'application les indique elle-même.

Pour éviter aux programmeurs de devoir régir l'intégralité de l'ordonnancement des tâches de leurs applications pour chaque machine utilisée, notre proposition est d'établir un dialogue entre l'environnement d'exécution et l'application pour aboutir automatiquement à un ordonnancement optimisé. L'application décrit l'organisation de ses tâches en regroupant, par exemple, celles travaillant sur les mêmes données (affinité mémoire). L'ordonnanceur du système peut alors exploiter ces indications pour adapter au mieux la répartition des tâches aux niveaux de hiérarchie qui composent la machine.
Bien sûr, un ordonnanceur « universel » qui donnerait d'excellents résultats avec uniquement ce type d'information reste à concevoir. Pour s'en approcher, nous offrons aux applications des possibilités supplémentaires en matière de « guidage » de l'ordonnancement. Nous proposons notamment des fonctionnalités permettant d'interroger le système à propos de la topologie de l'architecture sous-jacente. Le programmeur peut ainsi essayer et évaluer facilement différentes stratégies de regroupement de ses tâches. Il est même possible qu'il découvre des affinités qu'il n'aurait pas soupçonnées. Plus qu'un modèle d'ordonnancement, c'est donc une véritable plateforme d'expérimentation d'ordonnanceurs pour architectures multiprocesseurs que nous proposons.
Dans cet article, après avoir présenté les principales approches actuelles d'exploitation des machines hiérarchiques, nous proposerons deux nouveaux modèles pour décrire les tâches d'une application et la hiérarchie d'une machine, ainsi qu'un ordonnanceur qui en tire parti. Une sélection de détails d'implémentation sera alors présentée ainsi que des résultats d'évaluation en situation, avant de conclure.

2. Exploitation de machines hiérarchiques

Qu'elles soient simples SMP, NUMA, ou encore NUMA de multi-cores multi-threadés, les machines multiprocesseurs sont difficiles à exploiter de manière optimale. Différentes approches ont été envisagées.

2.1. Placement et ordonnancement prédéterminés

Pour une machine et une application données, il est parfois possible de déterminer un ordonnancement des tâches et un placement des données adaptés à la machine et ses niveaux de hiérarchie. En exigeant de l'environnement d'exécution cet ordonnancement et ce placement précis, on obtient alors des performances excellentes (voire optimales). Le solveur PaStiX [5] illustre particulièrement bien cette approche : il résout de (très) grands systèmes linéaires creux par une méthode directe en calculant d'abord un ordonnancement statique des calculs par blocs et des communications, par l'intermédiaire d'une simulation utilisant une modélisation des opérateurs BLAS et de la communication sur l'architecture cible.
Pour mettre en œuvre ces ordonnancements, de nombreux systèmes (AIX, LINUX, SOLARIS, WINDOWS,...) permettent de fixer les threads d'un processus sur des ensembles de processeurs ainsi que de préciser sur quel nœud mémoire les allocations de zones de données doivent être effectuées. Dans la mesure où la machine est dédiée à l'application, l'ordonnancement est maintenu totalement sous contrôle en fixant sur chaque processeur exactement un thread. Pour passer d'une tâche à une autre, on utilise alors des changements de contexte explicites, les threads étant utilisés comme simples conteneurs de fil d'exécution.

2.2. Placement et ordonnancement opportunistes

Étudiés depuis bientôt 20 ans, les algorithmes gloutons (appelés *Self-Scheduling* [6]) constituent des solutions dynamiques, souples et portables pour la parallélisation des boucles. Quelle que soit la machine sur laquelle l'application s'exécute, un algorithme de *Self-Scheduling* s'occupe de l'ordonnancement des threads et du placement des données. Les ordonnanceurs des systèmes d'exploitation actuels reposent sur ces algorithmes.
Le principe de base est l'utilisation d'une liste unique de tâches prêtes : l'ordonnanceur se contente de « piocher » dans celle-ci le prochain thread à ordonnancer, ce qui permet de répartir le travail entre les différents processeurs. Le dernier placement de chaque thread est mémorisé afin de le rejouer le plus possible, pour éviter les contre-performances dues aux défauts de cache. Ces techniques sont utilisées dans les systèmes d'exploitation LINUX 2.4 et WINDOWS 2000 [7]. Cependant, une unique liste de threads pour toute une machine constitue un goulet d'étranglement, en particulier lorsque la machine

possède de nombreux processeurs.
Pour éviter cette contention, les algorithmes *Guided Self-Scheduling* (GSS) [8] et *Trapezoid Self-Scheduling* (TSS) [9] font en sorte que chaque processeur s'approprie tout une partie du travail total lorsqu'il n'en a plus, au risque de provoquer d'éventuels déséquilibres. Les algorithmes *Affinity Scheduling* (AFS) [10] et *Locality-based Dynamic Scheduling* (LDS) [11] utilisent quant à eux des listes de tâches locales à chaque processeur. À court de travail, ce dernier en « vole » à un autre processeur, le plus chargé, par exemple. Ce dernier type d'algorithme est utilisé par les systèmes d'exploitation actuels (par exemple LINUX 2.6 [12], CELLULAR IRIX [13], FREEBSD 5.0 [14]) qui y ajoutent des éléments de rééquilibrage : tout processus créé est placé sur le processeur le moins chargé ; régulièrement, le processeur le plus chargé confie une partie de son fardeau au processeur le moins chargé.
Cependant, s'il y a de nombreux processeurs, et surtout pour les machines NUMA, ce n'est plus suffisant. WANG *et al*. proposent *Clustered Affinity Scheduling* (CAFS) [15] qui regroupe les p processeurs par groupes de $\sqrt{p}$. Lorsqu'un processeur est inoccupé, plutôt que de chercher à « voler » le travail du processeur le plus chargé de la machine, il cherche à « voler » celui du processeur le plus chargé de son groupe seulement, localisant ainsi les accès aux listes. De plus, en alignant les groupes sur les nœuds NUMA, on localise aussi le placement des données. Enfin, dans *Hierarchical Affinity Scheduling* (HAFS) (WANG *et al*. [16]), tout groupe entier inoccupé « vole » du travail au groupe le plus chargé. Cette dernière voie est en cours d'exploration pour le développement des systèmes d'exploitation tels que LINUX et FREEBSD.

2.3. Placement et ordonnancement négociés

Des approches intermédiaires entre ordonnancement prédéterminé et opportuniste existent. Les extensions de langage comme OPENMP [17], HPF (*High Performance Fortran*) [18] ou UPC (*Unified Parallel C*) [19] permettent en effet de programmer les machines parallèles à l'aide de simples indications. Une boucle `for` sera par exemple annotée pour que les différentes itérations soient automatiquement réparties entre différents threads, profitant ainsi des processeurs disponibles sur la machine. Une matrice de HPF pourra être annotée pour être automatiquement découpée en autant de blocs qu'il y a de processeurs, chaque processeur étant alors chargé d'effectuer les calculs concernant la partie qui lui a été attribuée.
C'est au compilateur que revient le travail de placement et d'ordonnancement. Pour cela, il ajoute le code déterminant l'environnement d'exécution (nombre de processeurs) et compile le programme de manière suffisamment générique pour pouvoir s'adapter aux différentes architectures parallèles. En particulier, il faudra gérer les threads nécessaires aux boucles parallélisées ou aux calculs répartis, voire gérer les échanges de données entre processeurs (cas des matrices réparties de HPF)... L'expressivité est cependant limitée : on obtient un programme où le parallélisme est surtout une succession de blocs « `Fork-Join` », le programmeur ne peut pas exprimer un parallélisme déséquilibré par exemple.
Il est aussi possible d'écrire des applications capables de s'adapter elles-mêmes à la machine supportant leur exécution. Les applications peuvent en effet prendre connaissance de leur environnement, les systèmes d'exploitations modernes fournissant une description complète de la machine les exécutant (voir les bibliothèques `lgroup` [20] pour SOLARIS ou `numa` [12] pour LINUX). L'application peut ainsi déterminer le nombre de processeurs (ce qui est commun), mais aussi obtenir la hiérarchie des nœuds NUMA, leurs nombres de processeurs et quantités de mémoire vive. De plus, ces mêmes systèmes d'exploitation permettent de déterminer la politique d'allocation mémoire (nœud mémoire précis, *first touch* ou *round robin*) et de fixer les threads sur des ensembles de processeurs (notion de *cpuset*). L'application contrôle alors à volonté (voire exactement) le placement des threads et de la mémoire, mais elle a alors l'entière responsabilité de la répartition des threads sur les processeurs.

2.4. Discussion

Les trois catégories dans lesquelles nous avons choisi de classer les approches existantes peuvent être caractérisées ainsi :

L'approche prédéterminée permet d'obtenir des performances excellentes. Elle n'est cependant portable que lorsqu'elle est appliquée aux problèmes réguliers, c'est-à-dire que leur traitement dépend de la structure des données et non pas des données elles-mêmes.

L'approche opportuniste passe bien à l'échelle, mais elle ne tient pas compte des affinités entre les

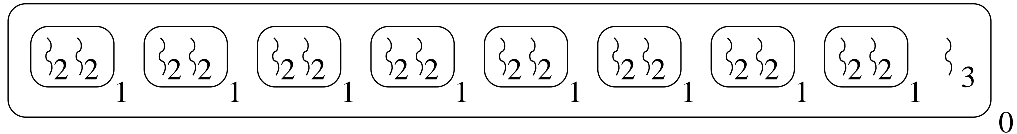


FIG. 1 – Exemple d'utilisation des bulles avec priorités : un ensemble de threads plus prioritaires que les bulles qui les contiennent et un thread très prioritaire.

tâches et n'obtient donc pas, en moyenne, d'excellentes performances.

L'approche négociée permet à l'application de s'adapter à la machine sous-jacente, mais demande, pour être flexible, l'écriture d'une partie de l'ordonnanceur.

Notre objectif est de proposer une nouvelle approche négociée qui, cependant, ne nécessite pas la réécriture d'un ordonnanceur. Pour cela nous allons donner aux programmeurs le moyen de décrire dynamiquement la structure du comportement de leur programme et utiliser cette description pour guider un ordonnanceur opportuniste.

3. Proposition : un ordonnanceur guidé par l'application

Notre approche repose sur la collaboration entre l'application et son environnement d'exécution.

3.1. Des bulles pour exprimer la structure d'une application

Nous demandons à l'application de modéliser l'organisation générale de ses threads au moyen d'ensembles imbriqués appelés **bulles** [1].

La figure 1 illustre une telle modélisation : l'application regroupe des threads de calcul par paires, et accompagne cette série de paire d'un thread de communication (les priorités seront détaillées plus loin).

La notion de bulle est à interpréter comme une *classe d'équivalence par rapport à une relation d'affinité donnée*, l'imbrication des bulles signifiant le *raffinement* de la relation par une autre relation. En effet, différentes relations d'affinités sont à considérer, citons par exemple :

Partage de données Il est profitable de rapprocher des threads travaillant sur les mêmes données afin de profiter des effets de cache ou du moins éviter qu'ils se retrouvent sur différents nœuds NUMA et soient alors pénalisés par le facteur NUMA.

Opérations collectives Il est intéressant d'optimiser l'ordonnancement des ensemble des threads concernés par une même opération collective telle une barrière de synchronisation qui permet de s'assurer que les threads concernés ont tous terminé la première partie d'un calcul avant d'entamer la seconde.

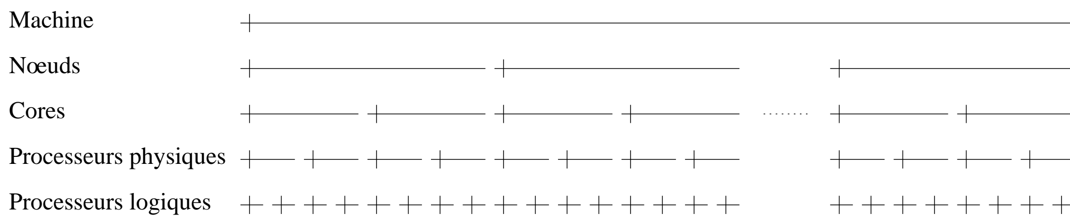
SMT Certains couples de threads sont à même d'exploiter très efficacement, sans interférer, la technologie du *Simultaneous Multi-Threading* lorsqu'ils sont exécutés en parallèle sur les deux processeurs logiques d'un même processeur physique.

D'autres relations d'équivalence sont possibles, on peut vouloir exprimer des qualités de parallélisme (voire de séquentialité), de préemption ou de priorité, par exemple.

Notons qu'au début des années 80, avec l'émergence des réseaux de machines multiprocesseurs, OUSTERHOUT [21] propose de regrouper les threads et données par affinités sous forme de *gangs*. Ceux-ci doivent contenir un nombre fixe de threads qui sont exécutés en parallèle sur une même machine du réseau : *Gang Scheduling*. Cependant des processeurs peuvent rester inoccupés car toute machine n'exécute qu'un seul gang à la fois même s'il est « petit ». FEITELSON *et al.* [22] proposent de hiérarchiser le contrôle des processeurs pour résoudre ce problème en éxécutant plusieurs gangs sur une même machine.

[1] De manière relativement analogue à certaines bibliothèques de communication telles que MPI qui demandent à l'application d'indiquer des *communicateurs* : les groupes de machines qui communiqueront ensemble.

(a) Une NUMA de multi-cores hyper-threadés.



(b) Modélisation par des listes de tâches.

FIG. 2 – Une machine très hiérarchique et sa modélisation.

Par ailleurs, le système SOLARIS permet aux applications de fixer leurs threads sur des *LWP* (*Light-Weight Processes*). Le système s'occupe de répartir les *LWP* sur les différents processeurs, tandis que la librairie de thread utilisateur peut, sur chaque *LWP*, ordonnancer les threads qui ont été fixés dessus.

3.2. Des listes de tâches pour coller à la structure de la puissance de calcul

D'après DANDAMUDI et CHENG [23], il est plus performant, en général, de hiérarchiser les listes de tâches prêtes que de les distribuer simplement entre processeurs. Aussi, des ordonnanceurs à deux niveaux de listes ont été développés [24, 25]. Qui plus est, ces ordonnanceurs facilitent la fixation de tâches sur un processeur donné. Nous avons repris ce point de vue, en le généralisant.

En effet, nous modélisons les machines hiérarchiques à l'aide d'une hiérarchie de listes de tâches prêtes. À chaque élément de chaque étage de la hiérarchie de la machine correspond (*bijectivement*) une liste de tâches. La figure 2 montre une machine hiérarchique et sa modélisation. La machine en entier, chaque nœud NUMA, chaque puce (*core*), chaque processeur physique SMT et chaque processeur logique possède ainsi une liste de tâches prêtes.

Cette identification entre élément physique et liste de tâches permet de déterminer l'espace d'exécution d'une tâche donnée : placée sur une liste associée à une puce physique, cette tâche pourra être exécutée par tout processeur de cette puce ; placée sur la liste globale elle pourra être exécutée par tout processeur de la machine.

Toute organisation peut être prise en compte. Il est ainsi aisé de modéliser un ensemble de machines

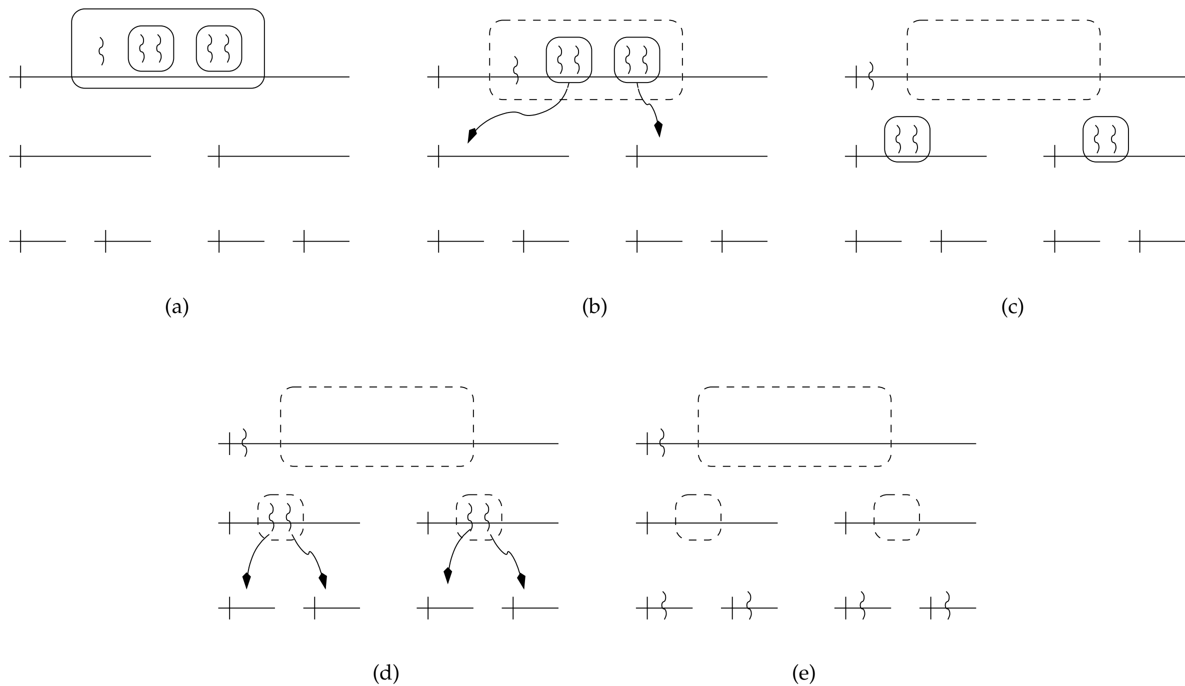


FIG. 3 – Évolution des bulles. (a) : La plus grande bulle a été placée sur la liste générale. (b) Elle éclate, libérant un thread (qui peut déjà être exécuté par n'importe quel processeur) et deux sous-bulles qui peuvent descendre dans la hiérarchie. (c) Descente effectuée. (d) Les deux sous-bulles éclatent, libérant chacune deux threads. (e) Les threads sont convenablement répartis et peuvent démarrer en parallèle.

NUMA reliées par un réseau à capacité d'adressage (tel que SCI [2] [27]) : il suffit simplement d'ajouter le niveau « réseau ». Bien entendu, suivre l'organisation physique de la machine n'est pas obligatoire. Par exemple, les *clusters* artificiels de processeurs introduits par CAFS (présenté en 2.2) peuvent eux aussi être modélisés.

3.3. Assemblage des deux modélisations : un ordonnanceur à bulles

Une fois les bulles créées par l'application, l'ordonnanceur de l'environnement d'exécution considère indifféremment les threads seuls et les bulles : ce sont des « tâches ». Il les fait alors évoluer pour les répartir sur la hiérarchie de listes de tâches.

3.3.1. Évolution des bulles

Comme l'illustre la figure 3, le rôle d'une bulle est d'encapsuler les tâches qu'elle contient pour les amener jusqu'au niveau où il sera intéressant de les ordonnancer. Pour cela, la bulle change de liste de tâches en descendant dans la hiérarchie jusqu'à arriver au niveau *voulu* (nous reviendrons sur ce terme). Là, elle « éclate », c'est-à-dire qu'elle libère les threads et bulles qu'elle contient pour les laisser s'exécuter (ou descendre à leur tour). Elle conserve tout de même l'inventaire de son contenu, pour une éventuelle reformation de la bulle (cf section 3.3.3).
Le problème central est bien sûr de spécificier le bon niveau d'éclatement d'une bulle (i.e. le niveau « *voulu* » mentionné précédemment).
À terme, lorsque nous aurons trouvé un ensemble d'heuristiques permettant de définir un ordonnanceur à bulle universel, il ne sera plus nécessaire de spécifier un tel paramètre. Mais pour l'heure, il s'agit de fournir une plate-forme d'expérimentation permettant de concevoir et de mettre au point des ordon-

[2] Un réseau SCI permet de définir des segments de mémoire partagées par plusieurs machines de façon transparente, la machine SEQUENT NUMA-Q [26] utilise cette technique.

nanceurs, donc ce paramètre est réglable par le concepteur de l'ordonnanceur. Dans l'état actuel, il peut ainsi utiliser ce levier pour privilégier plutôt les relations d'affinité entre tâches au risque de rendre plus difficile la réalisation de l'équilibrage de charge (niveaux d'éclatement bas) ou au contraire privilégier l'occupation des processeurs (niveaux d'éclatement hauts).

Ainsi, pour fixer le niveau d'éclatement d'une bulle, le concepteur doit donc indiquer pour chaque bulle son niveau virtuel d'éclatement sous forme d'un entier entre 1 et un certain n qu'il aura choisi. L'ordonnanceur répartit alors linéairement ces n niveaux virtuels sur les niveaux effectivement fournis par la machine, de telle façon que le niveau virtuel 1 corresponde au niveau réel « machine entière » et le niveau virtuel n au niveau réel « processeur logique ».

3.3.2. Notion de priorité

Nous avons choisi de permettre à l'application d'attacher à chaque bulle ou tâche une priorité représentée par un entier. Lorsqu'un processeur logique cherche une tâche à exécuter, il aura à parcourir, des plus *locales* (*i.e.* de niveau le plus bas) aux plus *globales*, les différentes listes qui le couvrent, à la recherche de la tâche de priorité maximale. Il devra alors exécuter cette tâche même si d'autres tâches moins prioritaires résident sur des listes plus locales.

La figure 1 montre un exemple mettant en œuvre les priorités. Dans cet exemple, les bulles contenant des threads de calcul sont moins prioritaires que les threads de calcul eux-mêmes. Une bulle ne sera donc éclatée que si les threads de calcul des bulles précédentes sont terminés, ou s'ils n'occupent pas tous les processeurs disponibles. Cela revient à effectuer un *Gang Scheduling* amélioré où tous les processeurs sont automatiquement occupés.

3.3.3. Reformation des bulles

Les bulles sont automatiquement réparties sur les différents niveaux et listes de tâches de la machine par l'ordonnanceur, répartissant ainsi les threads sur toute la machine en tâchant de respecter leurs affinités. Il se peut cependant que tout un groupe de threads ait en fait bien moins de travail à effectuer que d'autres et termine avant eux, laissant inoccupée toute la partie de la machine où ils s'exécutaient.

Pour *corriger* un tel déséquilibre, il est possible de reformer puis de remonter une ou plusieurs bulles. Un processeur sélectionnera une bulle ainsi reformée, la redescendra et l'éclatera alors de nouveau, adaptant ainsi la répartition du travail à la nouvelle situation de charge, tout en tâchant de respecter les informations d'affinités attachées à la bulle.

Pour *prévenir* l'apparition de déséquilibres, on peut même reformer régulièrement les bulles (analoguement à la préemption des threads par un système UNIX) : chaque bulle dispose d'un crédit de temps d'exécution (appelé *timeslice*) au bout duquel ses threads sont préemptés pour qu'elle soit reformée.

Dans le cas de l'exemple figure 3.3.2, on généralise le mécanisme de préemption au *Gang Scheduling* : à sa reformation une bulle est remise à la fin de la liste des tâches alors qu'une autre bulle est ouverte pour occuper les processeurs à son tour.

3.4. Discussion

Grâce aux bulles le programmeur a la capacité d'exprimer la structure de l'application ; ce faisant il extrait les caractères réguliers de celle-ci et guide son ordonnancement de façon *simple*, *portable* et *structurée*. Le rôle des différents processeurs et éléments de la machine n'étant pas prédéterminé, l'ordonnanceur conserve un degré de liberté et peut donc adopter une stratégie opportuniste pour placer et ordonnancer des ensembles cohérents de tâches. Prenant ainsi en compte le caractère irrégulier de l'application, l'ordonnanceur améliore significativement l'exploitation de la machine sous-jacente. Il faut cependant modérer nos propos : l'introduction d'éléments de rééquilibage correctifs et préventifs, comme toujours, peut avoir des effets pervers et conduire à des situations pathologiques (ping-pong d'une tâche, déplacement inutile des bulles juste avant leur terminaison, etc.)

4. Détails d'implémentation

Cette proposition a été implémentée au sein de la bibliothèque de threads utilisateur à deux niveaux MARCEL [28, 29]. Une interface simple de manipulation des bulles a été définie. La figure 4 illustre un exemple de formation et lancement d'une bulle contenant deux threads.

L'ordonnanceur disposant déjà d'une liste de threads prêts par processeur, l'intégration des bulles au

```
marcel_t thread1, thread2;
marcel_bubble_t bubble;

marcel_bubble_init(&bubble);
marcel_create_dontsched(&thread1, NULL, function1, parameter1);
marcel_create_dontsched(&thread2, NULL, function2, parameter2);
marcel_bubble_inserttask(&bubble, thread1);
marcel_wake_up_bubble(&bubble);
marcel_bubble_inserttask(&bubble, thread2);
```

FIG. 4 – Exemple de mise en place d'une bulle : on crée les threads sans les lancer avant de les insérer dans une même bulle. On peut lancer la bulle avant même d'avoir terminé d'insérer les threads

sein de la bibliothèque n'a pas nécessité la refonte complète des structures de données. Le code de l'ordonnanceur a par contre été adapté pour appliquer l'algorithme d'évolution et prendre en compte le mécanisme de priorité de la section 3. Cette implémentation est en fait assez délicate : si la hiérarchisation des listes de tâches permet de réduire les contentions, elle complexifie notablement le mécanisme de synchronisation.

La recherche de la tâche (thread ou bulle) de priorité maximale[3] par un processeur donné se fait en deux passes. La première passe détermine, sans verrouillage, la liste contenant la tâche la plus prioritaire. Cette liste et celle contenant le thread interrompu sont alors verrouillées[4]. Une deuxième passe de vérification est alors nécessaire pour être sûr que la liste sélectionnée possède encore une tâche de priorité maximale, un autre processeur ayant pu opérer sur cette liste. Lorsque la tâche sélectionnée est un thread, il est ordonnancé ; sinon, c'est une bulle que le processeur fait évoluer de façon apropriée (descente, éclatement). L'implémentation est finalement en $O(l)$ où l est le nombre de niveaux hiérarchiques de la machine.

La reformation d'une bulle est également une opération assez délicate. Pour replacer les tâches dans une bulle déterminée, il s'agit de retirer des listes toutes les tâches lui appartenant. Toutes sauf, bien entendu, les threads en cours d'exécution. Ces derniers se replaceront d'eux-mêmes dans la bulle lorsqu'ils exécuteront le code de l'ordonnanceur. Enfin, le dernier thread refermera la bulle et la remontera sur sa liste initiale (là où elle avait été libérée par la bulle qui la contenait).

5. Évaluation des performances

5.1. Coût de l'ordonnanceur et des bulles

Nous avons mesuré l'impact de notre implémentation sur les performances de la bibliothèque MARCEL sur un PENTIUM XEON cadencé à à 2.66GHz. Le parcours des listes en lui-même a un coût raisonnable et les temps d'exécution de notre ordonnanceur se comparent favorablement à ceux des bibliothèques de threads de LINUX LINUXTHREAD (noyau 2.4) et NPTL (noyau 2.6), voir figure 5.

D'autre part, créer et détruire une bulle contenant un thread n'est guère plus coûteux que créer et détruire ce thread seul : on passe de 3,3μs à 3,7μs de temps d'exécution pour les deux opérations[5].

On remarquera que, sur des exemples jouets de création récursive de threads tels que *sumtime* [29, 30], le coût induit par une création systématique de bulles exprimant la récursivité des créations est rapidement compensé par la localisation que ces indications apportent : lorsque peu de threads sont créés, une certaine contre-performance apparaît ; sur Bi-PENTIUM XEON (hyperthreadé), un gain se stabilisant entre 30 et 40% apparaît dès 16 threads ; sur NUMA 4×4 ITANIUM II, il est de 40% pour 32 threads et approche 80% à partir de 500 threads.

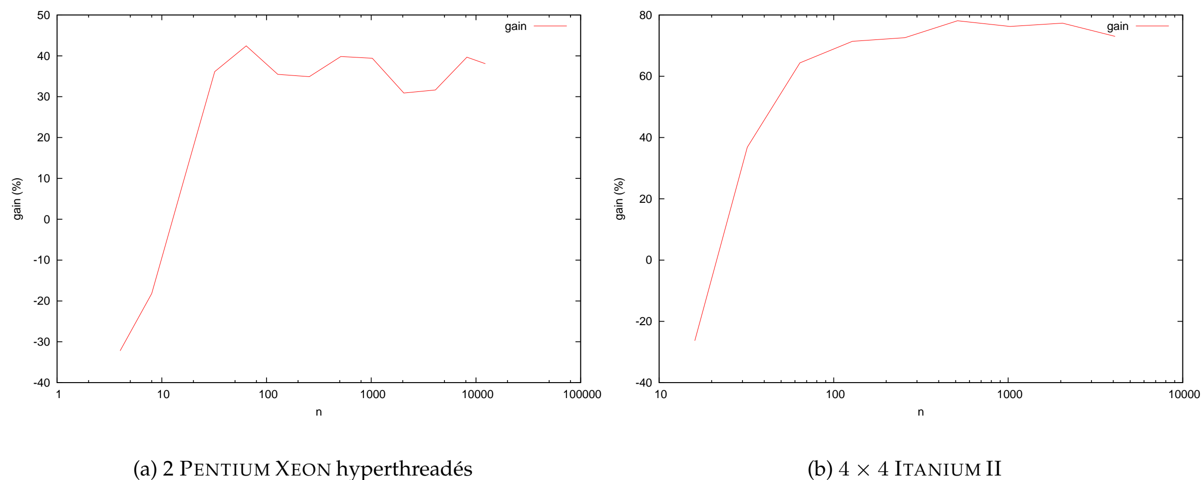
[3] Rappelons que pour un processeur donné sa tâche (thread ou bulle) la plus prioritaire est la première tâche de priorité maximale rencontrée en parcourant, de la plus locale au processeur à la plus générale, les listes « couvrant » le processeur en question.

[4] La convention adoptée pour verrouiller un ensemble de listes est de verrouiller en premier les listes de plus haut niveau, et pour un niveau donné, d'effectuer le verrouillage en suivant un ordre total prédéterminé.

[5] Ces opérations cumulées coûtent 42,3μs avec LINUXTHREAD et 7,8μs avec NPTL.

Changement de contexte seul	`setjmp`	7,1ns	∼ 19 cycles
	`longjmp`	5,6ns	∼ 15 cycles
MARCEL	`yield`	250ns	∼ 665 cycles
	`yield` + changement	398ns	∼ 1059 cycles
NPTL	`yield`	672ns	∼ 1790 cycles
	`yield` + changement	2150ns	∼ 5719 cycles
LINUXTHREAD	`yield`	672ns	∼ 1790 cycles
	`yield` + changement	1730ns	∼ 4600 cycles

FIG. 5 – Temps d'exécution de l'ordonnanceur MARCEL modifié pour le parcours de listes, avec changement de contexte (donc une certaine synchronisation) ou pas (parcours seul).



(a) 2 PENTIUM XEON hyperthreadés

(b) 4 × 4 ITANIUM II

FIG. 6 – Gain apporté par l'utilisation de bulles par *sumtime*.

5.2. Application concrète

Dans le cadre d'un travail de thèse au CEA DAM, Marc PÉRACHE utilise notre ordonnanceur dans **[31]** afin de comparer l'efficacité de différentes stratégies d'ordonnancement de l'exécution d'une application de calcul de conduction de chaleur. Les résultats obtenus sont présentés figure 7. La machine cible est une machine ccNUMA BULL NOVASCALE totalisant 16 processeurs et 64Go de mémoire. Ils sont répartis sur 4 nœuds NUMA de 4 processeurs et 4Go de mémoire. Pour un processeur donné, accéder à la mémoire du nœud où il est situé est de l'ordre de 3 fois plus rapide qu'accéder à la mémoire d'un autre nœud.

Dans la version MARCEL simple, le maillage est découpé en autant de bandes qu'il y a de processeurs et un ordonnancement opportuniste est utilisé. Pour obtenir un placement des threads et du maillage optimisé, il est possible de fixer de manière non portable les threads sur les processeurs, ce qui localise les accès mémoire : un thread et ses données restent sur un même nœud. On obtient alors de bien meilleures performances. Avec notre proposition, il suffit que l'application interroge MARCEL pour connaître le nombre de nœuds NUMA et de processeurs et constituer alors automatiquement des bulles selon la hiérarchie de la machine (ici 4 bulles de 4 threads), ce qui permet d'obtenir des performances très proches de la version « fixée ».

L'utilisation de bulles permet donc d'atteindre des performances comparables à celles obtenues en réglant « à la main » les placements de threads.

L'application CONDUCTION est pour l'instant un cas simple où la charge de calcul entre les bandes est équilibrée, l'utilisation de bulles lui permet simplement de s'adapter automatiquement à l'architecture

	temps	*speedup*
Séquentiel	250,2s	1
MARCEL simple	23,65s	10,58
MARCEL fixé	15,82s	15,82
MARCEL bulles	15,84s	15,80

FIG. 7 – Performances de l'application CONDUCTION selon l'approche. Le temps séquentiel sert de référence pour le *speedup*.

de la machine. Cette application devrait cependant prochainement être modifiée afin de bénéficier des méthodes de raffinement adaptatif (*AMR*) permettant d'augmenter la précision des calculs aux endroits intéressants. Ceci provoquera de gros déséquilibres en charge de calcul dans le maillage *au cours du calcul* et *selon les résultats obtenus*. Il sera intéressant de comparer les temps de développement et d'exécution des versions ordonnancées « à la main » d'une part, de façon opportuniste d'autre part, et enfin dynamiquement à l'aide de bulles.

6. Conclusion

Les machines multiprocesseurs deviennent de plus en plus hiérarchiques, ce qui rend l'ordonnancement des tâches sur de telles machines extrêmement complexe à maîtriser. Qui plus est, le véritable défi est d'aboutir à un ordonnanceur qui pourrait, à l'aide de directives d'ordonnancement portables, réaliser un « bon ordonnancement » des tâches sur une machine multiprocesseur à hiérarchie arbitraire.

Cet article présente un mécanisme original permettant de faire un pas dans cette direction : le modèle des bulles permet à une application d'exprimer les relations d'affinité entre les tâches d'une application (à des degrés variables) de manière portable. L'ordonnanceur peut ensuite utiliser ces directives pour le placement initial des threads, mais également lorsqu'il s'agit de rééquilibrer la charge de la machine.

Idéalement, l'ordonnanceur devrait pouvoir se contenter de ces seules informations pour effectuer ce travail. En pratique, l'élaboration d'un tel ordonnanceur est très difficile et nécessite de nombreuses expérimentations pour le mettre au point. En attendant, l'utilisateur peut utiliser des possibilités de guidage un peu plus strictes (spécification du niveau d'éclatement des bulles par exemple) pour expérimenter diverses stratégies. Nous pensons d'ailleurs que notre plateforme actuelle constitue une excellente base portable pour expérimenter différentes stratégies d'ordonnancement pour machines multiprocesseurs...

Les performances obtenues sur plusieurs jeux de tests sont très bonnes, bien meilleures que celles obtenues par simple ordonnancement opportuniste, et proches de celles obtenues par ordonnancement statique précalculé. Ces performances ont pu être observées sur différentes architectures (INTEL PC SMP, ITANIUM II NUMA).

Les perspectives ouvertes par ce travail sont nombreuses. À court terme, il est prévu de l'intégrer au sein de batteries de tests d'applications concrètes du CEA tournant sur machines très hiérarchiques, mettant ainsi à l'épreuve la puissance du mécanisme de bulles. Il sera alors souhaitable de développer des outils d'analyse basés sur un traçage de l'ordonnancement, pour être à même de vérifier et affiner de manière incrémentale les stratégies d'ordonnancement.

À plus long terme, l'objectif reste de fournir à l'application des moyens d'expression suffisament puissants et portables pour permettre un ordonnancement s'approchant de l'« optimal » quelle que soit l'architecture sous-jacente. Il serait par ailleurs utile de fournir à l'application des fonctions d'allocation mémoire plus expressives : préciser par exemple quelles tâches (ou tâches d'une bulle) utiliseront la zone allouée.

Bibliographie

1. HAGERSTEN (E.) et KOSTER (M.), « WildFire : A scalable path for SMPs », dans *The Fifth International Symposium on High Performance Computer Architecture*, janvier 1999. Sun Microsystems, Inc.
2. LAUDON (J.) et LENOSKI (D.), « The SGI Origin : A ccNUMA highly scalable server », dans *24th International Symposium on Computer Architecture*, p. 241–251, juin 1997. Silicon Graphics, Inc.
3. *Bull NovaScale servers*. `http://www.bull.com/novascale/`.
4. HENNESSY (J. L.) et PATTERSON (D. A.), *Computer Architecture : A Quantitative Approach*. Morgan Kaufman, 3e édition, 2003.
5. HÉNON (P.), RAMET (P.) et ROMAN (J.), « PaStiX : A parallel sparse direct solver based on a static scheduling for mixed 1d/2d block distributions », dans *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, p. 519–527. Springer-Verlag, janvier 2000.
6. TANG (P.) et YEW (P.-C.), « Processor self-scheduling for multiple nested parallel loops », dans *Proceedings 1986 International Conference on Parallel Processing*, p. 528–535, août 1986.
7. RUSSINOVICH (M.), « Inside the windows NT scheduler, Part 2 », *Windows IT Pro*, nº 303, juillet 1997.
8. POLYCHRONOPOULOS (C.) et KUCK (D.), « Guided self-scheduling : A practical scheduling scheme for parallel supercomputers », *Transactions on Computers*, vol. 36, nº 12, décembre 1987, p. 1425–1439.
9. TZEN (T.) et NI (L.), « Trapezoid self-scheduling : A practical scheduling scheme for parallel compilers », *Parallel and Distributed Systems*, vol. 4, nº 1, janvier 1993, p. 87–98.
10. MARKATOS (E.) et LEBLANC (T.), « Using processor affinity in loop scheduling on shared-memory multiprocessors », *Parallel and Distributed Systems*, vol. 5, nº 4, avril 1994, p. 379–400.
11. LI (H.), TANDRI (S.), STUMM (M.) et SEVCIK (K. C.), « Locality and loop scheduling on NUMA multiprocessors », dans *International Conference on Parallel Processing*, vol. II, p. 140–127, août 1993.
12. *Linux Scalability Effort*.
    `http://lse.sourceforge.net/`.
13. WHITNEY (S.), MCCALPIN (J.), BITAR (N.) *et al.*, « The SGI origin software environment and application performance », dans *COMPCON 97*, p. 165–170, San Jose, California, 1997. IEEE.
14. ROBERSON (J.), « ULE : A modern scheduler for FreeBSD ». Rapport technique, The FreeBSD Project, `jeff@FreeBSD.org`, 2003.
15. WANG (Y.-M.), WANG (H.-H.) et CHANG (R.-C.), « Clustered affinity scheduling on large-scale NUMA multiprocessors », *Systems Software*, vol. 39, 1997, p. 61–70.
16. WANG (Y.-M.), WANG (H.-H.) et CHANG (R.-C.), « Hierarchical loop scheduling for clustered NUMA machines », *Systems and Software*, vol. 55, 2000, p. 33–44.
17. MATTSON (T.) et EIGENMANN (R.), « OpenMP : An API for writing portable SMP application software », dans *SuperComputing 99 Conference*, novembre 1999.
18. SCHREIBER (R.), « An introduction to HPF », dans *The Data Parallel Programming Model : Foundations, HPF Realization, and Scientific Applications*, p. 27–44. Springer-Verlag, 1996.
19. *Unified Parallel C*. `http://upc.gwu.edu/`.
20. *Solaris Memory Placement Optimization (MPO)*.
    `http://iforce.sun.com/protected/solaris10/adoptionkit/tech/mpo/mpo_man.html`.
21. OUSTERHOUT (J. K.), « Scheduling techniques for concurrent systems », dans *Third International Conference on Distributed Computing Systems*, p. 22–30, octobre 1982.
22. FEITELSON (D. G.) et RUDOLPH (L.), « Evaluation of design choices for gang scheduling using distributed hierarchical control », *Parallel and Distributed Computing*, vol. 35, 1996, p. 18–34.
23. DANDAMUDI (S.) et CHENG (S.), « Performance impact of run queue organization and synchronization on large-scale NUMA multiprocessor systems », *Systems Architecture*, vol. 43, 1997, p. 491–511.
24. FUKUDA (A.), FUKIJI (R.) et KAI (H.), « Two-level processor scheduling for multiprogrammed NUMA multiprocessors », dans *Computer Software and Applications Conferences*, p. 343–351. IEEE, 1993. `fukuda@csce.kyushu-u.ac.jp`.
25. OGUMA (H.) et NAKAYAMA (Y.), « A scheduling mechanism for lock-free operation of a lightweight process library for smp computers », *Conference on Parallel and Distributed Systems*, juillet 2001, p. 235–242.
26. LOVETT (T. D.), CLAPP (R. M.) et SAFRANEK (R. J.), « NUMA-Q : An SCI-based enterprise server ». Rapport technique, Sequent Computer Systems Inc., 1996.

27. SOLUTIONS (D. I.), *The Dolphin SCI Interconnect*, février 1996. http://www.dolphinics.com/pdf/whitepaper/T-WhitePaper.pdf.
28. NAMYST (R.), *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. Thèse de doctorat, Univ. de Lille 1, janvier 1997.
29. DANJEAN (V.), *Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs*. Thèse de doctorat, École Normale Supérieure de Lyon, décembre 2004.
30. THIBAULT (S.), *Un ordonnanceur flexible pour machines multiprocesseurs hiérarchisées*. Rapport de stage DEA, École Normale Supérieure de Lyon, juillet 2004. http://perso.ens-lyon.fr/samuel.thibault/stage/3/rapport.ps.gz.
31. PÉRACHE (M.), « Nouveaux mécanismes au sein des ordonnanceurs de threads pour une implantation efficace des opérations collectives sur machines multiprocesseurs », dans *Rencontres francophones en Parallélisme, Architecture, Système et Composant (RenPar 16)*, mars 2005.